Introduction to Parallel & Distributed Programming

Lec 11—Memory Consistency (Causal, Processor), Synchronisation 1 — Flush & Atomic

Subodh Sharma | Feb 3, 2026

RECAP: Synchronisation

Types of Synchronisation Tools

- **Memory fences** (eg: `# pragma omp flush`, h/w memory fences like `mfence`)
- **Atomic Operations:** event should happen uninterrupted
  - **Test & set, Fetch & add, Compare & swap**
- **Critical sections, Lock, Mutexes:** Events should **NOT** happen together
- **Barriers:** Events should happen together
- **Wait, Condition variables:** event A should happen before event B

RECAP: Properties of Synchronisation

- Safety, Liveness
- Blocking
- Starvation-free, Deadlock-free, Lock-free, Wait-free

	Not lock-based Independent of Scheduler	Lock-based Depends on Scheduler
Everyone Progresses	Wait Free	Starvation Free
Someone Progresses	Lock Free	Deadlock Free

RECAP: The Flush Operation

- Flush directive performs two primary actions:
  - It forces the thread's temporary view of the variables to be written back to memory
  - It forces the thread to invalidate its local copy and reload vars from the memory
- Flush doesn't provide atomicity or mutual exclusion
  - It only ensures consistent visibility

RECAP: The Flush Operation

Thread A	Thread B
flagA = 1;	flagB = 1;
#pragma omp flush	#pragma omp flush
if (flagB == 0) {	if (flagA == 0) {
shared ++; ← mutual exclusion →	shared++;
}	}
flagA = 0;	flagB = 0;
#pragma omp flush	#pragma omp flush

RECAP: Atomic Operations

Test & Set

- **Test-and-Set (TAS):** Atomically reads a location and sets it to 1 and returns the old value
  - **Semantics:**

```
bool old_value = *location;
*location = true;
return old_value;
```

  - Can be used to implement Locks
  - **Limitations?**

RECAP: Atomic Operations

Fetch-and-Add

- **FAA -** Atomically adds a value to a memory location and returns the old value
- **Signature: T fetch_add(T* location, T increment)**
- **Semantics:**

```
T old_value = *location;
*location = old_value + increment;
return old_value;
```

- **Limitations:** No conditional update, Limited to addition, cache line contention, overflows-underflows

Atomic Operations

Compare & Swap

- **CAS:** Atomically compares a memory location to an expected value, and if they match, updates to a new value.
- **Signature:**

```
bool compare_and_swap(T* location, T* expected, T new_value)
```

- **Semantics:**

```
if (*location == *expected) {
    *location = new_value;
    return true;
} else {
    *expected = *location;  //
    return false;
}
```

Atomic Operations

Compare & Swap

- **CAS:** Atomically compares a memory location to an expected value, and if they match, updates to a new value.
- **Signature:**

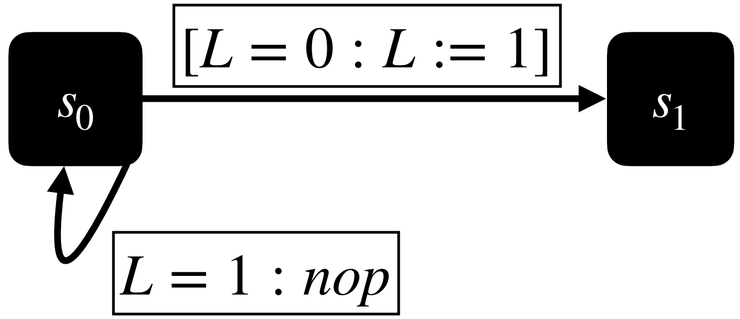
```
bool compare_and_swap(T* location, T* expected, T new_value)
```

- **Semantics:**

```
if (*location == *expected) {
    *location = new_value;
    return true;
} else {
    *expected = *location;  //
    return false;
}
```

- **Primary Use: Lock-free DS**

Implementing Lock: Using TAS

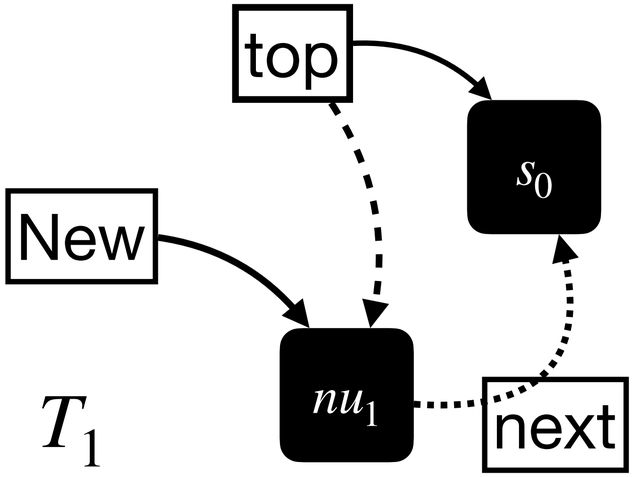


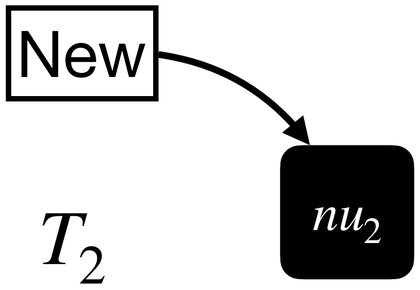

```
bool old_value = *location;
*location = true;
return old_value;
```

```
std::atomic_flag lockV = ATOMIC_FLAG_INIT
void lock(){
  while (lockV.test_and_set()){
    // lock was held — so keep spinning!
  }
}
                    void unlock(){
                      lockV.clear();
                    }
```

```
TASLock myLock;
void inc(){
  myLock.lock();
  shared_ctr ++;
  myLock.unlock();
}
```

Implementing Lock-free Stack: FAS & CAS






```
class LockFreeStack{

…

  void  push (T value){

    Node * new = new Node (value);

    Node * old_top = top.load();

    do{

      new->next = old_top;

    } while(!top.compare_and_exchange(old_top,
    new))

    size.fetch_add(1);

  }
```

RECAP: Synchronisation

Types of Synchronisation Tools

- **Memory fences** (eg: `# pragma omp flush`, h/w memory fences like `mfence`)
- **Atomic Operations:** event should happen uninterrupted
  - **Test & set, Fetch & add, Compare & swap**
- **Critical sections, Lock, Mutexes:** Events should **NOT** happen together
- **Barriers:** Events should happen together
- **Wait, Condition variables:** event A should happen before event B